HAL
archives-ouvertes.fr

Lambda Architecture pour une analyse à haute performance des données des réseaux sociaux

Annabelle Gillet, Eric Leclercq, Nadine Cullot

▶ **To cite this version:**

Annabelle Gillet, Eric Leclercq, Nadine Cullot. Lambda Architecture pour une analyse à haute performance des données des réseaux sociaux. INFormatique des ORganisations et Systèmes d'Information et de Décision, Jun 2019, Paris, France. hal-02413910

HAL Id: hal-02413910

https://hal-univ-bourgogne.archives-ouvertes.fr/hal-02413910

Submitted on 16 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Lambda Architecture pour une analyse à haute performance des données des réseaux sociaux

Annabelle Gillet, Éric Leclercq, Nadine Cullot

Laboratoire d'Informatique de Bourgogne - EA 7534
Univ. Bourgogne Franche-Comté
9, Avenue Alain Savary
F-21078 Dijon - France
annabelle.gillet@depinfo.u-bourgogne.fr, prénom.nom@u-bourgogne.fr

RÉSUMÉ. *Dans cet article, nous montrons comment une Lambda Architecture contribue à l'élaboration d'une plateforme de collecte et d'analyse en temps réel de données de Twitter. Après avoir présenté le contexte, détaillé les besoins et identifié les spécificités attendues, nous comparons les architectures Lambda et Kappa et nous dressons un état de l'art de l'utilisation de la Lambda Architecture dans différents domaines. Nous proposons ensuite une adaptation de la Lambda Architecture pour permettre le stockage de données dans un polystore et pour tenir compte de la multitude des types d'analyse à appliquer pour répondre à des projets de recherche en sciences sociales et en sciences de la communication dont les objectifs sont d'étudier la structure de la communication et la circulation du discours sur Twitter autour de sujets de société.*

ABSTRACT. *In this article, we show how a Lambda Architecture can contribute to the development of a platform for collecting and analyzing, in real-time, data from Twitter. After having presented the context, detailed the needs and identified the expected specificities, we compare the Lambda and Kappa architectures and we describe the state of the art on Lambda Architecture use in different domains. We propose an adaptation of the Lambda architecture to allow the storage of data in a polystore and to take into account different types of analysis to be carried out to answer researches in social sciences and communication sciences. In these projects the objectives are to study the structure of communication and the circulation of the speech on Twitter about some societal topics.*

MOTS-CLÉS : *Lambda Architecture, flux, polystore, données sociales*

KEYWORDS: *Lambda Architecture, streaming, polystore, social data*

1. Introduction

Les réseaux sociaux sont une source importante de données pouvant être exploitées dans de nombreux domaines comme le marketing, la politique, les sciences sociales. Les données produites font partie des Big Data et elles nécessitent des architectures logicielles spécifiques capables de prendre en compte des flux importants, de les stocker et de les analyser (Kambatla *et al*., 2014 ; Singh, Reddy, 2015). Ces problématiques sont regroupées sous les termes *High Performance Data Analysis* (HPDA) et *Data Intensive High Performance Computing*. Les données des réseaux sociaux sont riches mais leur sémantique est complexe et elles contiennent de nombreuses caractéristiques latentes. Pour effectuer une analyse détaillée de ces données, il est nécessaire d'avoir recours à plusieurs algorithmes comme la détection de communautés, la détection d'événements, la recherche d'utilisateurs influents, etc. Les résultats des analyses peuvent être ensuite exploités pour éclairer une question de recherche, élaborer une théorie, un modèle qui seront ensuite validés sur d'autres données sociales.

En fonction des observations que l'on souhaite réaliser ou des questions auxquelles on cherche à répondre, plusieurs algorithmes sont mis en œuvre et ils travaillent sur différentes modélisations des données. Par exemple, on peut vouloir trouver des variations des comportements des utilisateurs au fil du temps, auquel cas les séries temporelles sont bien adaptées pour représenter les données. Ensuite, sur une période identifiée, on peut regrouper des utilisateurs en communautés, afin d'avoir une vision plus précise de l'organisation et des interactions entre les communautés d'utilisateurs. Ainsi, plusieurs algorithmes sont appliqués séparément ou conjointement et ils fournissent des informations à différents niveaux de granularité (macroscopique, mésoscopique et microscopique). De plus, ils peuvent être exécutés en temps réel sur un flux ou en temps différé sur une plus grande quantité de données. Dans cet article, nous traitons plus particulièrement du patron Lambda Architecture (Marz, 2011) intégrant un traitement des flux en temps réel et des analyses en temps différé. Nous spécialisons l'architecture pour une plateforme de collecte et d'analyse de données de Twitter. Nous testons l'architecture avec différents jeux de données précédemment collectés.

L'article est organisé comme suit, la section 2 présente le contexte et décrit les principes des architectures de traitement de données à haute performance. La section 3 est un état de l'art sur l'utilisation des Lambda Architectures, la section 4 détaille notre proposition et la section 5 présente les résultats d'une série d'expériences permettant de valider les différents composants de l'architecture. La section 6 conclut l'article et dresse les perspectives dégagées par ce travail.

2. Contexte et problématique

Nous nous plaçons dans le cadre de projets interdisciplinaires développés avec des chercheurs en sciences sociales et en sciences de la communication dont l'objectif est d'étudier la structure de la communication et la circulation des discours sur Twitter (Basaille *et al*., 2017). Les thématiques abordées sont les questions de société comme les problèmes environnementaux, le changement climatique ainsi que les discours au-

tour de l'alimentation intégrant les comportements alimentaires, les crises sanitaires, etc. Dans ces domaines, les chercheurs en sciences sociales souhaitent détecter des messages viraux, l'émergence de hashtags populaires ou de topics c'est-à-dire des groupes de hashtags utilisés ensemble et structurant les discours. Il veulent aussi obtenir des informations sur les communautés d'utilisateurs ayant des comportements similaires et suivre leur évolution.

Afin d'effectuer des analyses, il faut tout d'abord pouvoir collecter les données et être capable d'absorber un flux important [1] de plusieurs milliers de messages par seconde pour constituer un jeu de données se rapprochant le plus possible de la réalité. Twitter met à disposition des développeurs des moyens pour récupérer les données publiées par leurs utilisateurs, à travers deux API différentes : *search* et *stream*. L'API *stream* est limitée à 1% du trafic global par machine connectée. Ce volume peut être très important si plusieurs machines sont utilisées en parallèle (500.10^6 tweets sont publiés quotidiennement). De plus, les tweets contiennent de nombreuses informations qu'il faut extraire comme l'utilisateur émetteur, la date d'émission, les hashtags utilisés, les utilisateurs mentionnés, la localisation, les URL citées, etc. En conséquence, une plateforme d'analyse à haute performance doit être capable d'absorber, en temps réel, un flux important de tweets, d'effectuer des analyses elles aussi en temps réel et de stocker les données en vue d'analyses plus complexes réalisées en temps différé. Avant de pouvoir utiliser les données collectées, il faut bien souvent les nettoyer, transformer les données brutes dans un modèle plus adapté aux algorithmes utilisés. Afin de limiter l'utilisation des processus ETL (Extract-Load-Transform) très consommateurs en temps de développement, les *polystores* proposent différents modes de stockage unifiés soit par le langage, soit par un modèle commun (Leclercq, Savonnet, 2018). Ils prennent ainsi en compte des données de type graphes attributaires, matricielles, relationelles et documents [2]. Par ailleurs, dans les domaines d'application qui nous intéressent, il est nécessaire de pouvoir suivre en permanence l'évolution des tendances, de détecter des événements. Ces analyses en temps réel peuvent être réalisées par des algorithmes de *streaming* (Gama, 2012 ; Silva *et al.*, 2013 ; McGregor, 2014) nommés algorithmes de fouille de flux de données; en ligne ou encore incrémentaux.

2.1. Principe de la Lambda Architecture

La Lambda Architecture est un patron d'architecture logicielle décrit par Nathan Marz (Marz, 2011), qu'il détaillera plus tard dans un ouvrage plus complet (Marz, Warren, 2015). Cette architecture permet de traiter les données massives en temps réel et par lots de manière simultanée. Les trois V d'origine des Big Data concernant le Volume, la Variété et la Vélocité ont été complétés par IBM avec la Véracité puis ensuite

1. Le flux moyen moyen mondial est de 6 000 tweets environ par seconde, avec de rares pics dépassant plus de 8 000 tweets par seconde sur des événements médiatisés comme les MTV Video Music awards (source http://www.internetlivestats.com/twitter-statistics/)

2. Utiles par exemple pour stocker le contenu des tweets et le contenu Web des URL référencées dans les tweets.

par SAS qui a introduit la Variabilité et enfin Oracle qui a ajouté la Valeur (Gandomi, Haider, 2015). La Lambda Architecture se concentre sur quelques V caractéristiques, la Vélocité, le Volume et dans une moindre mesure sur la Valeur. L'architecture se compose de trois couches (ou *layers*) : la *Batch layer*, la *Serving layer* et la *Speed layer* (figure 1). Le principe général est le suivant : la *Batch layer* permet de fournir une vue exacte sur les données, mais avec un temps de traitement ou de transformation important. La *Serving layer* met ensuite ces vues à disposition des utilisateurs, avec pour objectif d'optimiser le temps d'accès aux données. La *Speed layer* sert à compenser ce temps de traitement, en proposant un traitement des données en temps réel.

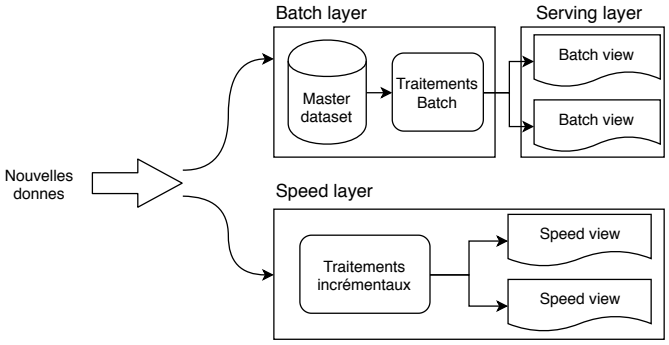


Figure 1. Principe général de la Lambda Architecture

La *Batch layer* a deux rôles principaux : 1) stocker les nouvelles données, et 2) réaliser des calculs, des transformations de modèles sur ces données. Le stockage des données brutes ne se fait pas sous la forme d'états mais sous la forme d'une suite d'événements ou de messages qui permettent d'obtenir un état actualisé des différents éléments lorsque ces évènements sont traités dans leur ensemble. La *Batch layer* stocke toutes les données (messages) qu'elle reçoit dans le *master dataset*, c'est-à-dire un entrepôt permanent de données brutes. Les calculs et les transformations de modèles sont opérés à partir du *master dataset*, souvent de manière distribuée afin de garantir le passage à l'échelle.

La *Serving layer* se charge ensuite de récupérer les résultats de ces traitements, et les met à la disposition des utilisateurs. Le but étant d'optimiser le temps d'accès, la modélisation des données est réalisée dans ce sens (utilisation des index, schéma non normalisé, etc.).

La *Speed layer* s'occupe de traiter les données en temps réel, au moyen d'algorithmes qui travaillent de manière incrémentale. Elle rend disponible les nouvelles données qui n'ont pas encore été traitées par la *Batch layer*. La performance prime donc sur l'exactitude (ou la véracité) des traitements réalisés, cette contrainte étant laissée à la charge de la *Batch layer*.

Les erreurs d'exactitude qui peuvent survenir lors du traitement de la *Speed layer* sont corrigées lorsque la *Batch layer* termine son traitement sur les mêmes données. Réciproquement, la lenteur de la *Batch layer* est bien compensée par la rapidité de la *Speed layer*, qui permet de voir un résultat des traitements en temps réel mais éventuellement avec des imprécisions.

La force de la Lambda Architecture vient donc de sa capacité à traiter des données en quantité importante et avec un flux soutenu, en compensant la latence des traitements *Batch* par des traitements sur des flux par la *Speed Layer*. Toutefois, notre domaine d'application induit une variation dans la construction de la Lambda Architecture. Nous devons satisfaire deux priorités : 1) préparer les données pour des analyses dont les algorithmes ne sont pas nécessairement connus *a priori* et 2) pouvoir calculer en temps réel des indicateurs macroscopiques. Dans la littérature la Lambda Architecture est critiquée car les traitements sont dupliqués dans les couches *Batch* et *Speed*, mais en réalité ce n'est pas valable pour tous les domaines d'application et encore moins pour notre cas d'utilisation.

2.2. Lambda, Kappa, et autres architectures, discussion

La Kappa Architecture est un autre patron qui adopte une vision différente de la Lambda Architecture. Elle peut être vue comme une simplification dans le sens ou elle considère que tous les traitements opèrent sur des flux (*everything is a stream*). Comme (Kreps, 2014) le décrit dans son article, plutôt que d'avoir une couche *Batch* et une couche *Speed*, il est possible de conserver uniquement une seule couche *Speed* et d'organiser les flux. Pour ce faire, les données doivent être conservées sous la forme d'une suite de messages (*logs*). S'il faut relancer un traitement sur les données, il suffit d'exécuter un autre traitement *Speed* en parallèle du principal, qui contient les modifications à apporter, et d'arrêter le premier lorsque le nouveau l'a rattrapé. Dans ce contexte Apache Kafka [3] est une solution utilisée majoritairement dans des architectures avec une composante temps réel. Bien que la durée de stockage (rétention) ou le volume des données soient limités, il est tout même possible de traiter les données depuis le début en les stockant en plus dans un système de fichiers distribué comme Apache Hadoop HDFS [4], pour ensuite les réinjecter dans des files de messages.

À titre de comparaison, les points forts de la Lambda Architecture sont : une conservation des données brutes qui permet de les retraiter si besoin, une solution flexible indépendante des technologies et une adaptation fine des outils d'analyse aux demandes des clients. Il s'agit cependant d'une architecture assez complexe à mettre en œuvre. Les points forts de la Kappa Architecture sont l'unification du principe de traitement par flux, sa simplicité et sa relative indépendance vis-à-vis des technologies. Cependant la Kappa Architecture n'est pas aussi flexible que la Lambda Architecture du point de vue de l'organisation des données. L'architecture SMACK (*Spark-Mesos-*

3. https://kafka.apache.org/
4. https://hadoop.apache.org/

Akka-Cassandra-Kafka) est une autre proposition d'architecture, figée du point de vue des outils, même si elle est souvent présentée comme concurrente des Kappa et Lambda Architecture, il s'agit plutôt d'une pile de technologies (*data analytics stack*).

En conclusion, la flexibilité de la Lambda Architecture doit être mise en perspective du haut niveau de technicité requis pour la mise en œuvre des différentes solutions techniques qu'elle intègre. Dans notre cas, il s'agit de la solution la plus pertinente au regard des différents types d'analyses à déployer sur des graphes, des séries temporelles, des données relationnelles ou des données textuelles. Une proposition dans ce sens a été élaborée dans l'article (Fernandez *et al*., 2015) pour LinkedIn avec l'architecture Liquid dont le principe fondateur est un chemin à faible latence (*Low Latency Path*). L'architecture Liquid recueille moins d'attention que la Lambda Architecture. En effet, cette dernière insiste sur une capacité d'analyse temps réel très séduisante alors que le patron d'architecture est très général et délègue aux composants logiciels retenus pour l'implémentation le soin d'assurer la prise en charge des flux et des traitements.

3. État de l'art

Avant de présenter quelques exemples de projets appliqués, nous décrivons plusieurs travaux qui permettent d'identifier les spécificités des Lambda Architectures.

Dans (Mishne *et al*., 2013) les auteurs relatent d'expériences d'architecture pour la correction orthographique et la suggestion de recherche en temps réel de tweets. Ces travaux montrent les limites des architectures s'appuyant uniquement sur Hadoop et ils sont les précurseurs de la Lambda Architecture. L'article (Lin, 2017) met en avant les concessions qu'implique l'adoption d'une Lambda Architecture. En effet, le mode de fonctionnement de la *Speed layer* et de la *Batch layer* sont différents, mais ils doivent pourtant produire des résultats similaires. Cela entraîne une complexité lors du développement et des évolutions de l'architecture. Dans (Feick *et al*., 2018) les auteurs décrivent et comparent les architectures Lambda et Kappa par rapport au théorème CAP et présentent de manière synthétique des outils d'implémentation pour les différentes couches. Ils relatent d'une expérimentation avec des données de Twitter avec un flux maximum de 26 000 tweets par jour et 2 300 par heure.

Du point de vue des domaines d'applications, on distingue deux catégories de projets : des instanciations de la Lambda Architecture pour des domaines précis et des travaux autour de la construction de plateformes génériques par assemblage de solutions logicielles open source.

Dans (Munshi, Mohamed, 2018), les auteurs présentent une implémentation de la Lambda Architecture appliquée aux traitements des données issues des réseaux électriques. La résistance aux pannes, le temps de réponse rapide, le passage à l'échelle et la flexibilité de ce type d'architecture sont les paramètres qui ont motivés leur choix d'utiliser la Lambda Architecture. Hadoop HDFS est utilisé pour implanter un *Data-Lake* et traiter de la diversité des données (capteurs, images, vidéos). La couche

d'interrogation, séparée de la couche d'analyse, intègre les technologies SparkSQL [5], Hive [6] et Impala [7]. Lee et Lin (2017) spécifient une Lambda Architecture pour construire un système de recommandation de restaurants. Leur apport est l'utilisation d'Apache Mesos [8] pour abstraire les composants matériels de la plateforme, faciliter son déploiement et sa mise à l'échelle. Les technologies utilisées sont Kafka pour la *Speed Layer* et les traitements temps réel, Hadoop HDFS pour le stockage des données brutes, et Spark pour la *Batch Layer*. Les performances de l'architecture sont testées avec le jeu de données MoviesLens 20M [9] et différentes configurations matérielles allant jusqu'à exploiter 32 coeurs CPU et 96Go RAM. Cependant, le volume des données et les flux sont trop peu importants pour fournir des expériences significatives.

RADStack (Yang *et al*., 2017) est une plateforme *open source* pour produire des analyses interactives, implémentant le principe de la Lambda Architecture et utilisant les briques logicielles Kafka, Samza [10], Hadoop et Druid [11]. Kiran *et al*. (2015) s'intéressent aux performances que peut offrir la Lambda Architecture, afin de minimiser les coûts qui peuvent être associés au déploiement des applications dans le *cloud*. Leur approche est mise en pratique avec le déploiement de leur architecture sur Amazon AWS, afin d'analyser des données issues de capteurs. Pal *et al*. (2018) proposent une évolution de la Lambda Architecture en intégrant un système multi-agents pour des applications de commerce électronique.

Comme Liquid (Fernandez *et al*., 2015) discutée précédemment, le système S-Store (Meehan *et al*., 2016) partage des similarités avec notre proposition en intégrant le polystore BigDAWG. Le rôle de S-Store est de nettoyer les données et de les transformer en vue de leur ingestion par le polystore. L'article présente un *proof-of-concept* avec des données des benchmarks TCP-C et TCP-DI [12]. Plutôt que d'écrire ses résultats dans un fichier, puis d'insérer le contenu du fichier en base de données comme un ETL standard, S-Store fait passer les données de processus en processus afin de leur appliquer une transformation jusqu'à leur insertion en base de données. L'intégration dans une Lambda Architecture est évoquée dans les perspectives mais dans l'état actuel la proposition est plutôt un ETL traitant des flux en temps réel.

4. L'architecture Hydre

L'architecture Hydre que nous proposons s'appuie sur la Lamba Architecture et profite de ses avantages, tout en l'adaptant afin de mieux répondre aux besoins de notre contexte de collecte, gestion et analyse des données issues de Twitter.

---

5. https://spark.apache.org/sql/
6. https://hive.apache.org/
7. https://impala.apache.org/
8. http://mesos.apache.org/
9. https://grouplens.org/datasets/movielens/20m/
10. http://samza.apache.org/
11. http://druid.io/
12. http://www.tpc.org/

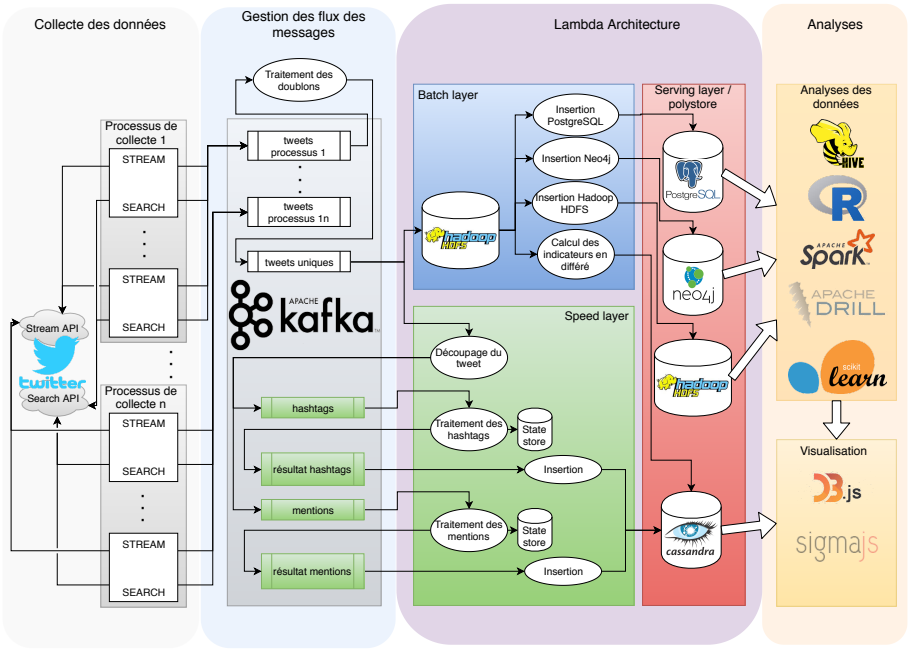


Figure 2. Hydre, Lambda Architecture détaillée

4.1. Architecture générale

La figure 2 présente l'architecture Hydre globale. Elle est majoritairement développée à l'aide du langage Scala [13], et agrège différents composants : 1) un système de collecte des données, faisant appel aux API de Twitter pour alimenter 2) un composant de gestion des flux des messages, qui utilise Kafka. L'extraction des messages ainsi collectés constitue notre point d'entrée dans la Lambda Architecture, et cette extraction est donc réalisée en parallèle par les deux couches : 3) la *Speed layer*, qui nous permet de calculer les séries temporelles sur les éléments importants par tranches d'une heure, comme des hashtags, des mentions, etc., et 4) la *Batch layer* qui enregistre d'abord les données brutes dans le *master dataset* implémenté avec Hadoop HDFS, puis réalise différents traitements sur ces données (comme recalculer les séries temporelles), et la *Serving layer* se charge d'insérer les données dans le stockage polystore. Ces données pourront ensuite être analysées et visualisées à l'aide de différents outils.

13. https://www.scala-lang.org/

4.2. La collecte des données

Afin de collecter des tweets, Twitter met à disposition deux API : l'API *search* [14] et l'API *stream* [15]. Bien que le but de ces API soit le même, leur fonctionnement est très différent. En effet, la première permet de collecter des tweets ayant été produits au maximum 7 jours plus tôt et qui correspondent à la requête qui a été fournie à l'API, tandis que la deuxième permet d'enregistrer une requête auprès de Twitter, qui renverra ensuite tous les tweets nouvellement produits correspondant aux critères spécifiés. Les critères des requêtes permettent de filtrer les tweets sur des mots-clés, des hashtags, des utilisateurs, etc. Les deux API nous fournissent les tweets au format JSON. Dans leur version gratuite, ces API imposent toutefois quelques limitations par compte utilisé et par adresse IP. En ce qui concerne l'API *search*, il est seulement possible d'envoyer 180 requêtes par créneaux de 15 minutes, en récupérant 100 tweets maximum à chaque fois. L'API *stream*, quant à elle, limite le nombre de tweets captés par l'application à 1% du trafic total de Twitter par machine connectée, et si les tweets ne sont pas consommés assez rapidement par l'application, la connexion avec Twitter peut être coupée. Pour communiquer avec ces API, nous utilisons la librairie Twitter4j [16].

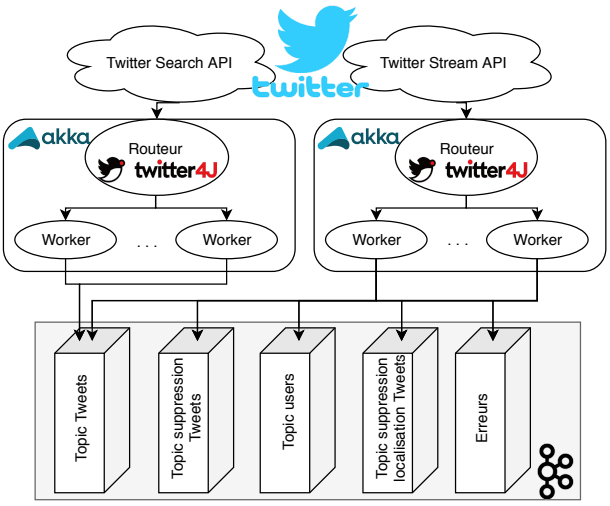


Figure 3. Collecte de tweets et articulation avec Kafka

Pour gérer les tweets que nous récoltons, nous utilisons des acteurs Akka [17] disponibles avec Scala. Akka est une implémentation de l'*Actor Model*, qui permet de

14. https://developer.twitter.com/en/docs/tweets/search/overview
15. https://developer.twitter.com/en/docs/tweets/filter-realtime/overview
16. http://twitter4j.org/en/index.html
17. https://akka.io/

réaliser des applications distribuées à base d'échanges de messages. Cela nous permet de paralléliser le traitement des tweets, de produire des messages de type (K, V) où K est la clé et V la valeur (le contenu du tweet). Si la rapidité n'est pas un critère primordial pour l'API *search* aux vues de ses limitations, cela devient un problème plus contraignant pour l'API *stream* dans le cas où le flux capté est très important. Nous utilisons la fonctionnalité *Routing* d'Akka, qui permet de définir un certain nombre de *Workers* auxquels le routeur transmet les messages qu'il reçoit pour qu'ils puissent les traiter (figure 3). Le nombre de *Workers* peut être fixe, comme c'est le cas pour notre traitement à la réception des tweets provenant de l'API *search*, ou il peut être adaptable dynamiquement au volume de messages que le routeur reçoit, ce que nous utilisons pour traiter les tweets provenant de l'API *stream*. L'architecture permet de traiter plusieurs corpus simultanément, chaque corpus dispose d'un ou plusieurs mécanismes de collecte en fonction de la volumétrie de tweets attendue pour le corpus (figure 2, partie grisée).

4.3. Système de gestion des flux

Le système Kafka est utilisé pour servir d'intermédiaire entre la partie collecte et les couches *Speed* et *Batch*. Il permet de créer des *topics* dans lesquels les producteurs peuvent poster des messages. Mais il permet aussi de former plusieurs groupes de consommateurs qui consultent les messages des *topics* à leur rythme et de manière indépendante (figure 2). Les messages sont alors conservés pendant un temps défini (délai de rétention) ou jusqu'à ce qu'ils atteignent un certain volume. Chaque topic a un usage particulier : gestion des messages tweets, profils utilisateurs, indicateurs de suppression, messages d'erreur.

Étant donné la configuration de la collecte, il est possible de recevoir un même tweet plusieurs fois. Que ce soit parce qu'un même mot-clé nous a permis de récupérer le tweet avec l'API *stream* puis avec l'API *search*, ou parce que des mots-clés différents utilisés sur des machines distinctes ciblent un même tweet. Bien que Kafka propose un mécanisme de *log compaction*, c'est-à-dire que pour une même clé de message seul le dernier message est conservé, cela n'est pas adapté à notre situation. En effet, si l'on prend en compte un tweet récupéré par l'API *stream*, et le même tweet qui peut être récupéré par l'API *search* jusqu'à 7 jours plus tard, il est fortement probable que le premier tweet ait déjà été traité avant qu'il ne soit remplacé par le deuxième, ce qui entraînerait un traitement en doublon. Nous développons donc un mécanisme spécifique pour le traitement des doublons en sortie de Kafka à l'aide de Kafka Streams. À la suite de ce traitement, les tweets uniques sont redirigés dans un topic, dans lequel les processus *Speed* et *Batch* vont puiser leurs données. Grâce à ce fonctionnement, nous pouvons alimenter les parties *Speed* et *Batch* avec des données uniques, et nous pouvons également récupérer des statistiques sur la collecte qui nous permettent de savoir combien de fois un tweet a été collecté, depuis quelle API et avec quelle requête. Ces éléments pourraient ensuite être utilisés dans une évolution de l'architecture, afin d'optimiser la répartition des mots-clés dans les requêtes utilisées par les machines de collecte pour grouper ceux qui apparaissent souvent ensemble.

4.4. La couche Speed

Le rôle principal de la *Speed layer* est, dans notre cas, de produire des indicateurs macroscopiques sur une collecte en cours et détecter des événements, comme les hashtags les plus utilisés, leur fréquence, ou les mentions faites d'un utilisateur. Ces indicateurs prennent la forme de séries temporelles par tranches de temps d'une heure. Cette couche est développée à l'aide de Kafka Streams, et les résultats sont insérés dans une base de données Cassandra.

Le fonctionnement est le suivant (figure 2 cadre et rectangles verts), un tweet est d'abord traité par un processus qui se charge de séparer les différentes parties (les hashtags, les mentions, etc.). Ces parties sont envoyées dans différents topics Kafka dans le but d'être traitées chacune par un processus spécifique. Les processus sont associés à un *state store* de Kafka, qui leur permettent de garder un état en mémoire. Nous nous en servons pour compter combien de fois un même élément a été rencontré sur une plage de temps donnée. Le compte mis à jour est ensuite envoyé à nouveau dans un topic, puis réceptionné par Cassandra qui se charge de mettre les données à jour.

Séparer les traitements de cette manière autorise une grande souplesse dans les évolutions de la couche *Speed*. En effet, il suffit juste d'adapter le processus de découpage pour extraire les nouveaux éléments qui pourraient nous intéresser, et chaque processus de traitement étant indépendant, il peut être ajouté, modifié ou retiré sans impacter le reste de la couche.

4.5. La couche Batch et la couche Serving

Lorsque de nouveaux tweets sont collectés et envoyés via Kafka, le processus *Batch* les récupère pour les ajouter dans le *master dataset*. Afin de remplir cette tâche, nous utilisons Hadoop HDFS, qui est un framework permettant de distribuer de larges volumes de données sur un cluster et d'y effectuer des traitements en utilisant le paradigme *map-reduce*. L'ajout de machines au cluster permet un passage à l'échelle efficace, en augmentant à la fois la capacité de calcul et de stockage. De nombreux autres outils constituent cet eco-système et permettent d'abstraire les mécanismes proposés par Hadoop.

Deux types de processus sont développés. Un processus *Batch* se charge de recalculer les séries temporelles produites par le processus *Speed* afin de les remplacer. Un ensemble de processus est chargé de récupérer les nouvelles données et, pour chaque base de données composant notre polystore, de les insérer tout en respectant le schéma défini. De cette manière, les données que nous récoltons sont transformées dans différents modèles afin de pouvoir ensuite être exploitées par les différents algorithmes d'analyse. Différents types de bases de données sont utilisés afin de correspondre aux besoins des algorithmes : Cassandra pour les séries temporelles, Noe4j pour les modèles graphes et PostgreSQL pour stocker les données des tweets comme la géolocalisation ou les informations des utilisateurs. Grâce à l'organisation des éléments

de notre architecture, il est aisé de rajouter une autre base de données, de modifier ou ajouter un schéma d'insertion si cela est nécessaire. Il suffit en effet de modifier ou de rajouter un processus d'insertion dans les parties *Batch* et *Serving* afin que ces changements soient pris en compte.

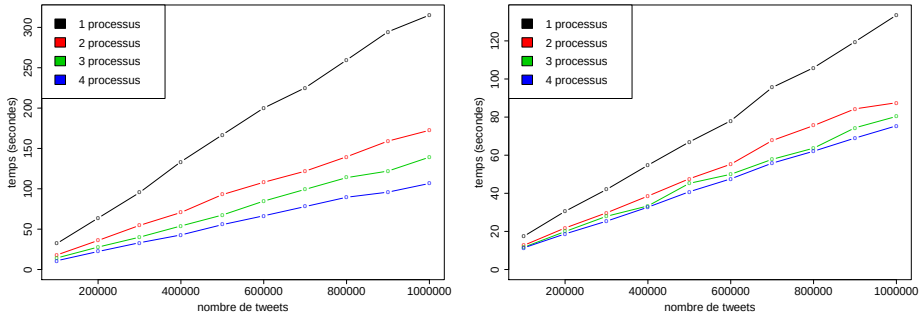
5. Retours d'expériences

Afin de valider la capacité de l'architecture Hydre à supporter la charge imposée à ses composants, différentes expériences sont réalisées. Elles sont définies pour les 3 niveaux de l'architecture : 1) au niveau du système de production des messages, pour mesurer la capacité de Kafka à ingérer des flux importants, 2) au niveau de la couche *Batch*, de la récupération des messages depuis Kafka pour enregistrer les tweets dans HDFS, à l'extraction des données d'HDFS pour les insérer dans les systèmes de stockage du polystore, 3) au niveau de la *Speed layer* pour mesurer sa capacité à produire des séries temporelles et à les matérialiser dans Cassandra. La configuration matérielle sur laquelle ont été effectués les tests est la suivante : 1) un serveur Dell R710 (Intel Xeon CPU E5-2650 v2 @ 2.60GHz, 16 cœurs, 128Go RAM) pour la couche de collecte et Kafka (3 brokers dans des conteneurs Docker) ; 2) un serveur Dell R940 (Intel(R) Xeon(R) CPU E7-4820 v2 @ 2.00GHz, 32 cœurs, 256Go RAM) pour les couches *Batch*, *Speed* et *Serving* ; 3) un cluster Hadoop 20 nœuds *data nodes* et 2 *name nodes*, 184 cœurs, 1040Go RAM, 70To de stockage HDFS.

Ingestion de Kafka des messages. Kafka est un composant principal de l'architecture. Il garde tous les tweets obtenus avec les processus de collecte pour qu'ils puissent être consommés par les couches *Speed* et *Batch*. Afin d'évaluer la capacité de cette partie à supporter un flux important, des producteurs Kafka envoient un certain nombre de tweets, et le temps est mesuré depuis l'envoi du premier message jusqu'à la réception du dernier. Pour réduire l'impact des éléments extérieurs sur l'expérience, les tweets sont obtenus à partir d'un modèle, et seul l'identifiant de chaque tweet est produit aléatoirement (figure 4a).

Enregistrement des tweets dans HDFS. Dans la partie *Batch*, le premier évènement lors de l'arrivée d'un message est son enregistrement sous forme de donnée brute dans Hadoop. Nous produisons donc en amont un certain nombre de messages uniques dans Kafka, que nous consommons ensuite. Nous mesurons le temps pris pour que le nombre de tweets défini soit enregistré dans Hadoop (figure 4b).

Insertion dans un schéma relationnel normalisé ou non. Une fois que les tweets sont enregistrés dans Hadoop, un consommateur de la partie *Batch* peut ensuite les récupérer pour les enregistrer dans une base de données. C'est ce que nous testons dans cette expérience, en extrayant les informations du tweet de son format JSON et en l'insérant dans une base de données PostgreSQL sous une forme non normalisée (une seule table) (figure 5a) et normalisée (11 tables avec index) (figure 5b). Nous réalisons aussi l'expérience pour l'insertion dans Neo4j (3 types de noeuds et 4 types de relations) (figure 5c). Pour ces trois cas, nous mesurons le temps que le processus met pour extraire les tweets d'Hadoop et les insérer en base de données.



(a) Ingestion dans Kafka

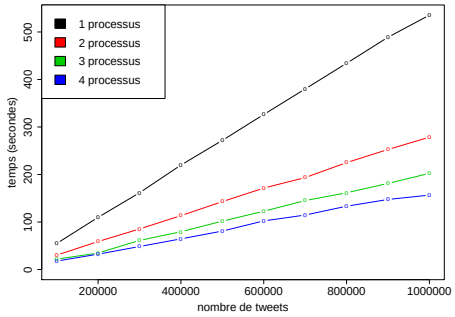
(b) Consommation et enregistrement HDFS

Figure 4. Production des messages (Akka, Kafka) et enregistrement (Kafka, HDFS)

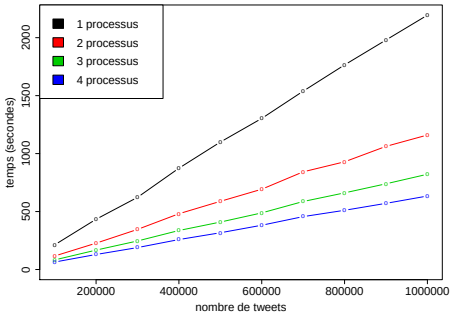
Mise à jour de séries temporelles. Cette expérience nous sert à mesurer le temps nécessaire à la couche *Speed* pour mettre à jour la série temporelle concernant les hashtags des tweets. Le traitement se compose de 3 étapes réalisées avec Kafka Streams : 1) extraire les hashtags du tweet, 2) compter le nombre d'occurrences par heure, 3) insérer ce résultat dans Cassandra. Pour cette expérience, les messages Kafka sont d'abord produits, puis le traitement *Speed* est lancé. Le temps est mesuré depuis l'extraction des hashtags du premier message, jusqu'à la dernière insertion dans Cassandra (figure 6a). La même expérience est réalisée pour mettre à jour les séries temporelles, mais cette fois du côté de la partie *Batch* : les données sont extraites d'Hadoop, une requête est envoyée sur Cassandra pour obtenir le nombre d'occurrences pour le hashtag en train d'être traité, puis ce nombre d'occurrences est incrémenté (figure 6b).

L'étude des performances des composants principaux d'Hydre est synthétisée dans le tableau suivant qui donne le nombre de tweets par seconde traitable par chaque composant et démontre la capacité de l'architecture à supporter le flux moyen de Twitter.

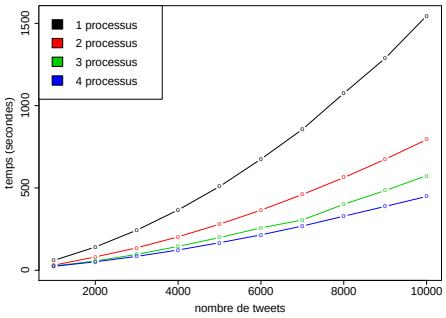
Interprétation des résultats. On observe que les différents composants traitent les flux de tweets d'une manière plutôt linéaire, et se prêtent assez bien à la parallélisation (figures 4, 5 et 6). Les performances de la partie *Speed* restent très bonnes par rapport à celles de la partie *Batch* pour ce qui concerne le calcul des séries temporelles (d'autant plus qu'il faut compter le temps d'enregistrement dans Hadoop pour la partie *Batch*). Neo4j, quant à lui, présente d'importants problèmes de performances, et même s'il se parallélise bien, il montre une allure exponentielle lors d'opérations d'insertion. Cela nous amène à étudier d'autres bases de données orientées graphes afin de le remplacer par un système plus adapté à nos besoins. Nous indiquions plus haut que le flux moyen de Twitter est de 6 000 tweets/s et que nous récupérions au maximum 1% de ce flux par machine, les résultats que nous avons obtenus (tableau 1) nous permettent de traiter sereinement les quantités de tweets que nous nous attendons à recevoir.



(a) Insertion dans PostgreSQL (schéma non normalisé)



(b) Insertion dans PostgreSQL (schéma normalisé)



(c) Insertion dans Neo4j

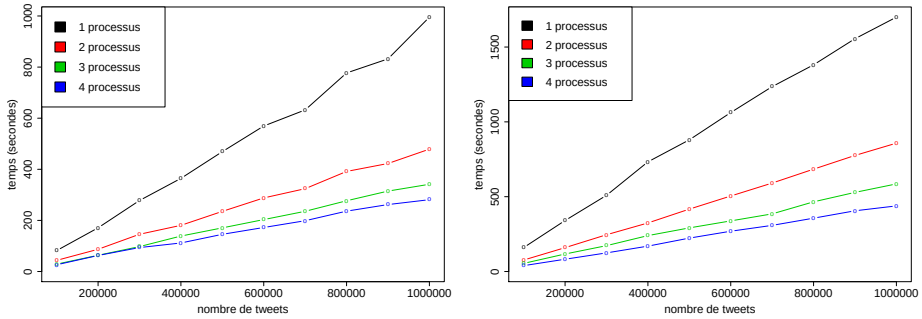
Figure 5. Lecture depuis Hadoop HDFS et insertion dans PostgreSQL et Neo4j

Tableau 1. Nombre de tweets par seconde admissible pour chaque composant

Nombre de processus	Ingestion Kafka	Stockage HDFS	Insertion non normalisé	Insertion normalisé	Insertion Neo4j	Série temporelle (Speed)	Série temporelle (Batch)
1	3 082	7 181	1 840	460	10	1 091	579
2	5 615	10 194	3 508	855	19	2 149	1208
3	7 227	11 364	4 977	1 210	26	3 010	1739
4	9 128	12 017	6 089	1 563	30	3 459	2324

6. Conclusion

Nous avons présenté une adaptation de la Lambda Architecture afin de répondre aux spécificités des analyses de données pour des projets en sciences sociales pour



(a) Production de la série temporelle hashtags dans Cassandra (couche *Speed*)

(b) Production de la série temporelle hashtags dans Cassandra (partie *Batch*)

Figure 6. Production de séries temporelles

lesquels les algorithmes à appliquer ne sont pas nécessairement connus *a priori*. Les composants sont architecturés autour de Kafka et permettent d'insérer des données dans un polystore et de calculer des indicateurs en temps réel pour détecter des événements. Nous avons ainsi développé et testé un bus de messages à faible latence, capable de supporter le flux des tweets produits quotidiennement. Notre implémentation permet de rajouter ou modifier simplement des traitements dans sa partie *Batch* ou *Speed*. Nous prévoyons d'améliorer les services de la couche *Speed* afin de détecter en temps réel des précurseurs d'évènements, des messages viraux et des robots. Sur un plan plus théorique nous allons réaliser une typologie des cas d'utilisation des données sociales pour les analyses, tester les performances des systèmes de stockage et intégrer le modèle TDM (Leclercq, Savonnet, 2018). Sur le plan technique, nous envisageons d'utiliser Kafka schema registry pour gérer les évolutions des schémas des messages, KSQL et/ou Apache Samza pour implanter les autres algorithmes de flux (Noghabi *et al.*, 2017).

Bibliographie

Basaille I., Kirgizov S., Leclercq É., Savonnet M., Cullot N., Grison T. *et al.* (2017). Un observatoire pour la modélisation et l'analyse des réseaux multi-relationnels. *Document numérique*, vol. 20, n° 1, p. 101–135.

Feick M., Kleer N., Kohn M. (2018). Fundamentals of Real-Time Data Processing Architectures Lambda and Kappa. In *Lecture Notes In Informatics (LNI)*, p. 1-12.

Fernandez R. C., Pietzuch P. R., Kreps J., Narkhede N., Rao J., Koshy J. *et al.* (2015). Liquid: Unifying Nearline and Offline Big Data Integration. In *Conference on Innovative Data System Research (CIDR'15)*.

Gama J. (2012). A survey on learning from data streams: current and future trends. *Progress in Artificial Intelligence*, vol. 1, n° 1, p. 45–55.

Gandomi A., Haider M. (2015). Beyond the hype: Big Data concepts, methods, and analytics. *International Journal of Information Management*, vol. 35, n° 2, p. 137–144.

Kambatla K., Kollias G., Kumar V., Grama A. (2014). Trends in Big Data Analytics. *Journal of Parallel and Distributed Computing*, vol. 74, n° 7, p. 2561–2573.

Kiran M., Murphy P., Monga I., Dugan J., Baveja S. S. (2015). Lambda architecture for cost-effective batch and speed big data processing. In *IEEE International Conference on Big Data*, p. 2785–2792.

Kreps J. (2014). Questioning the Lambda Architecture. *O'Reilly RADAR,online article, July*. Consulté sur https://www.oreilly.com/ideas/questioning-the-lambda-architecture

Leclercq É., Savonnet M. (2018). A tensor based data model for polystore: An application to social networks data. In *Proceedings of the 22nd International Database Engineering & Applications Symposium (IDEAS)*, p. 110–118.

Lee C.-H., Lin C.-Y. (2017). Implementation of Lambda Architecture: A Restaurant Recommender System over Apache Mesos. In *IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, p. 979–985.

Lin J. (2017). The Lambda and the Kappa. *IEEE Internet Computing*, vol. 21, n° 5, p. 60–66.

Marz N. (2011). *How to beat the cap theorem.* Consulté sur http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

Marz N., Warren J. (2015). *Big Data: Principles and best practices of scalable real-time data systems*. Manning.

McGregor A. (2014). Graph stream algorithms: a survey. *ACM SIGMOD Record*, vol. 43, n° 1, p. 9–20.

Meehan J., Zdonik S., Tian S., Tian Y., Tatbul N., Dziedzic A. *et al*. (2016). Integrating real-time and batch processing in a polystore. In *High performance extreme computing conference (hpec), 2016 ieee*, p. 1–7.

Mishne G., Dalton J., Li Z., Sharma A., Lin J. (2013). Fast data in the era of big data: Twitter's real-time related query suggestion architecture. In *Proceedings of the 2013 acm sigmod international conference on management of data*, p. 1147–1158.

Munshi A. A., Mohamed Y. A.-R. I. (2018). Data lake lambda architecture for smart grids big data analytics. *IEEE Access*, vol. 6, p. 40463–40471.

Noghabi S. A., Paramasivam K., Pan Y., Ramesh N., Bringhurst J., Gupta I. *et al*. (2017). Samza: stateful scalable stream processing at linkedin. *VLDB Endowment*, vol. 10, n° 12, p. 1634–1645.

Pal G., Li G., Atkinson K. (2018). Multi-agent big-data lambda architecture model for e-commerce analytics. *Data*, vol. 3, n° 4, p. 58.

Silva J. A., Faria E. R., Barros R. C., Hruschka E. R., De Carvalho A. C., Gama J. (2013). Data stream clustering: A survey. *ACM Computing Surveys (CSUR)*, vol. 46, n° 1, p. 13.

Singh D., Reddy C. K. (2015). A survey on platforms for big data analytics. *Journal of big data*, vol. 2, n° 1, p. 8.

Yang F., Merlino G., Ray N., Léauté X., Gupta H., Tschetter E. (2017). The RADStack: Open source lambda architecture for interactive analytics. In *Proceedings of the 50th hawaii international conference on system sciences.*